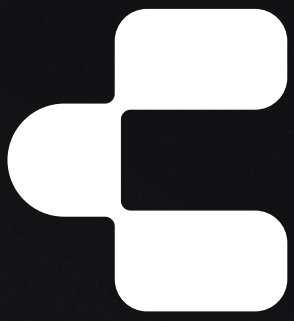
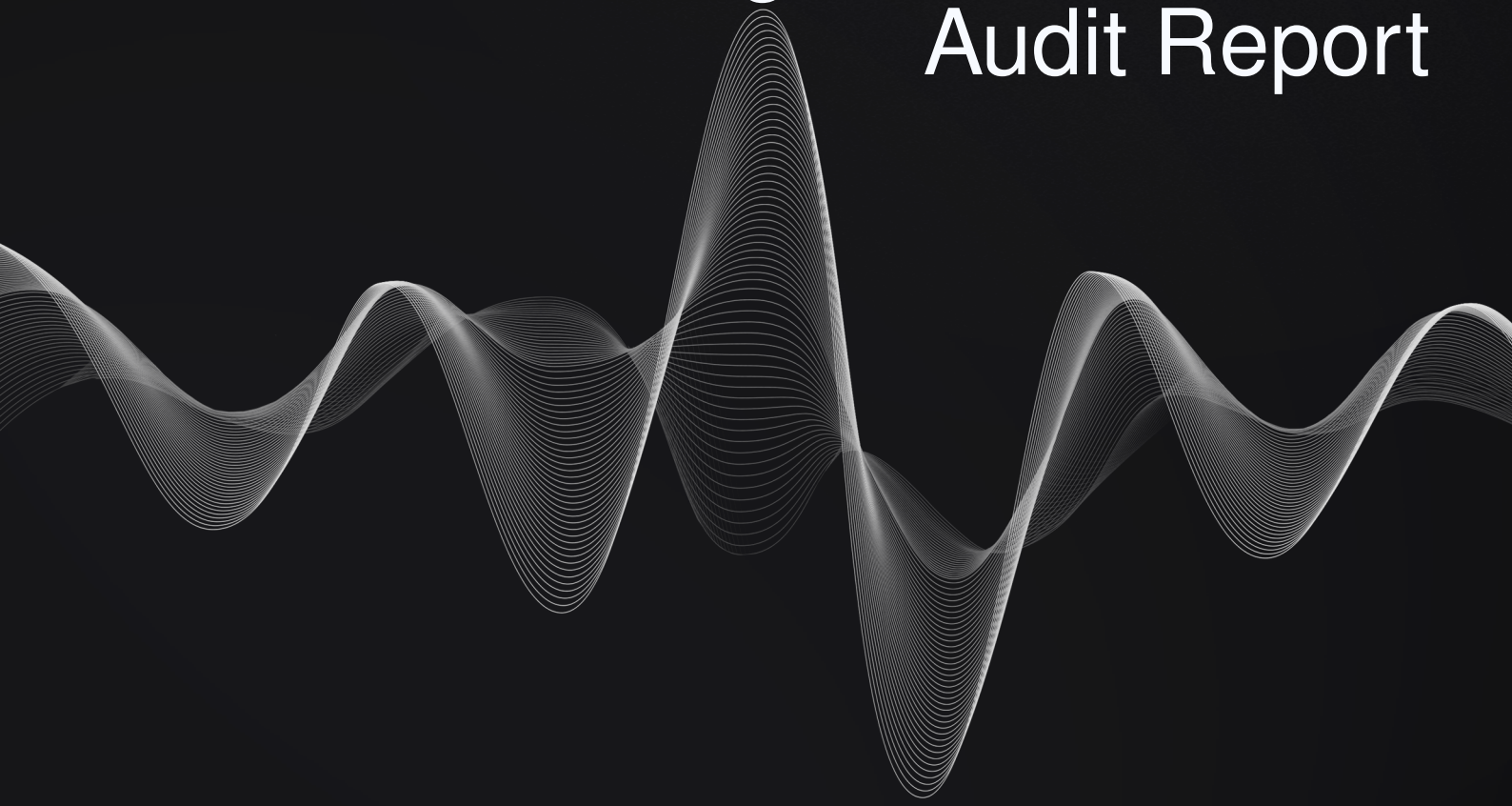
RESONANCE



ConsumerFi

Merkle And Investor Vesting Smart Contract Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.1)

Audit_Report_COFI-MIV_FINAL_21

Nov 5, 2025	v0.1	João Simões: Initial draft
Nov 6, 2025	v0.2	João Simões: Added findings
Nov 7, 2025	v1.0	Charles Dray: Approved
Nov 21, 2025	v1.1	João Simões: Reviewed findings
Dec 4, 2025	v2.0	Charles Dray: Finalized
Dec 4, 2025	v2.1	Charles Dray: Published

Points of Contact	Christopher Tan	ConsumerFi	director@flamefoundation.xyz
	Charles Dray	Resonance	charles@resonance.security
Testing Team	João Simões	Resonance	joao@resonance.security
	Ilan Abitbol	Resonance	ilan@resonance.security
	Luis Arroyo	Resonance	luis.arroyo@resonance.security

Copyright and Disclaimer

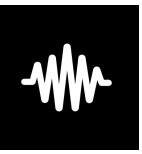
Contents

Executive Summary

ConsumerFi contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between October 31, 2025 and November 7, 2025. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 6 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide ConsumerFi with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.



System Overview

The ConsumerFi smart contracts comprise an Investor Vesting Distributor and Merkle Distributor contracts that together form a two-part NEAR-based token distribution framework designed to manage complex investor allocations securely and transparently. Both contracts are built around the NEP-141 token standard and enable administrators to preload tokens, define vesting or distribution logic, and allow beneficiaries to claim tokens progressively or under specific conditions. These systems support token vesting and claim automation for both on-chain and off-chain allocation models, ensuring flexibility for investor programs, community distributions, and incentive schemes.

The Investor Vesting Distributor implements a time-based vesting mechanism where administrators configure vesting profiles—such as cliffs followed by linear unlock schedules—and assign token allocations to investors. Tokens are held in the contract and become claimable as vesting conditions mature, either by investors directly or by an admin on their behalf. The contract supports standard lifecycle operations, including initialization, funding, investor assignment, claims, and withdrawal of unallocated tokens. Its logic enforces linear vesting after configurable cliffs and ensures that allocations unlock deterministically based on elapsed time since the token generation event.

The Merkle Distributor extends this capability by introducing an off-chain allocation and proof-based claiming system using a Merkle tree. Administrators generate allocation proofs off-chain and deploy them to the contract alongside vesting group configurations. Beneficiaries can then claim using their proof, selecting one of several supported claim strategies: standard linear vesting, lock options with reward multipliers, instant partial claims, or stake-required programs that unlock tokens proportionally to staked collateral (e.g., xCFI). This architecture enables scalable, gas-efficient distribution of large token sets with verifiable integrity while supporting flexible incentive models and lock mechanics managed entirely on-chain.

Repository Coverage and Quality

Code	Tests	Documentation
		
6 / 10	7 / 10	9 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code does not follow development best practices and does not make use of standard libraries, and language guides, but does make use of known patterns. It is not easily readable but uses the latest stable version of relevant components. Overall, **code quality is average**.
- Integration tests are included. The tests cover functional requirements. Code coverage is undetermined. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `ymc182/investor-claim-contract/src`
- Hash: 796ce49f08580cf75382f0b99a9670bbb47a6f56
- Repository: `ymc182/merkle-claim-contract/src`
- Hash: 52336d228ff6c9e2fb7668455535cbe3acc37cf3

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - JavaScript NEAR

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for JavaScript NEAR audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Race conditions caused by asynchronous cross-contract calls
- Frontrunning attacks
- Storage staking
- Potentially problematic storage layout patterns
- Manual state rollbacks in callbacks
- Access control issues

- Denial of service
- Inaccurate business logic implementations
- Unoptimized Gas usage
- Arithmetic issues
- Client code interfacing

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

Impact \ Likelihood	Very Likely	Likely	Unlikely
Strong	Critical	High	Medium
Medium	High	Medium	Low
Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	Usage Of Native Collections For Smart Contract State		Resolved
RES-02	Improper Implementation Of Dynamic Storage Management Calculation		Resolved
RES-03	Missing Validation Of Account Ids		Resolved
RES-04	Missing Validation Of Input Parameters Against Default Or Undefined Values		Resolved
RES-05	Missing storage_withdraw() And storage_unregister()		Resolved
RES-06	Unprotected Initialization Function Can Be Frontrun		Acknowledged
RES-07	Missing Validation Of Duplicate investors		Resolved
RES-08	Redundant Contract State Validation		Acknowledged
RES-09	JavaScript Contract Missing Schema		Acknowledged
RES-10	Redundant Code On init()		Acknowledged
RES-11	Redundant Code On assertSelf()		Acknowledged
RES-12	Unused Functions		Resolved
RES-13	Redundant Code On set_distribution() And set_merkle_root()		Resolved

Usage Of Native Collections For Smart Contract State

High | RES-COFI-MIV01 | Denial of Service | Resolved

Code Section

- investor-claim-contract/src/investor_claim_contract.ts
- merkle-claim-contract/src/contract.ts

Description

The contract stores critical on-chain state using native JavaScript objects, e.g. `Record`, instead of NEAR SDK collections. Native collections are not stored under predictable prefixes, and, as such, require full serialization and deserialization of the entire contract state on every call, which may require the insertion of a big dataset, leading to excessive storage reads/writes and potential out-of-gas failures as data grows.

It should be noted that this issue becomes more concerning due to the fact that NEAR's JavaScript SDK serializes the state into JSON objects, that are considerably more expensive than Borsh.

Recommendation

It is recommended to replace all native collections declared in the state with their relevant counterparts in NEAR SDK collections, using appropriate key prefixes for each logical dataset.

Status

The issue has been fixed in 77a14857b295b6e4a75dea151462a8eb8b07a73d.

Improper Implementation Of Dynamic Storage Management Calculation

High | RES-COFI-MIV02 | Code Quality | Resolved

Code Section

- merkle-claim-contract/src/contract.ts

Description

Due to the architecture of the NEAR blockchain based on payment of used storage, it is a best practice to implement storage staking mechanisms to ensure each user pays for the storage they use.

The smart contract makes use of dynamic structures represented as collections, however, it does not implement a proper storage staking mechanism. Malicious users could perform attacks such as the "Million Deposits" attack that could drain the contract of its funds.

The smart contract requires a storage deposit of `0.0125 N` per account, as per the constant `MIN_ACCOUNT_STORAGE_DEPOSIT`.

Given that a user that interacts with the protocol requires multiple collections to be inserted into and updated, fixed estimations may lead to both situations of under and overfunding, which can harm both the protocol and its users.

It should be noted that this issue is especially concerning due to the fact that the same account ID can participate in multiple distributions, therefore requiring further magnitudes of byte usage, when a single storage deposit was made.

Recommendation

It is recommended to dynamically measure the maximum amount of bytes used by a user of the protocol and require the respective NEAR value, and eventually refund storage differences (per NEP-145). An example of a dynamically calculated storage management implementation can be seen on https://github.com/near/near-sdk-js/blob/develop/packages/near-contract-standards/lib/fungible_token/core_impl.js#L35 .

Status

The issue has been fixed in 77a14857b295b6e4a75dea151462a8eb8b07a73d.

Missing Validation Of Account Ids

Medium | RES-COFI-MIV03 | Data Validation | Resolved

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L59-L88
- investor-claim-contract/src/investor_claim_contract.ts#L104
- investor-claim-contract/src/investor_claim_contract.ts#L140
- merkle-claim-contract/src/contract.ts#L68-L113
- merkle-claim-contract/src/contract.ts#L163
- merkle-claim-contract/src/contract.ts#L481-L483
- merkle-claim-contract/src/contract.ts#L640
- merkle-claim-contract/src/contract.ts#L795

Description

The smart contract does not validate NEAR Account Ids that are served as input parameters and uses simple strings instead. This can lead to unnecessary errors for users during runtime or, business logic flaws that can ultimately lead to worse consequences, such as draining protocol funds.

Recommendation

It is recommended to use the class `AccountId` and the function `validateAccountId()`, both present in NEAR's JavaScript SDK to ensure provided account ids are properly validated.

Status

The issue has been fixed in 77a14857b295b6e4a75dea151462a8eb8b07a73d.

Missing Validation Of Input Parameters Against Default Or Undefined Values

Medium | RES-COFI-MIV04 | Data Validation | Resolved

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L84-L86
- investor-claim-contract/src/investor_claim_contract.ts#L208
- merkle-claim-contract/src/contract.ts#L101
- merkle-claim-contract/src/contract.ts#L640
- merkle-claim-contract/src/contract.ts#L755-L759

Description

Throughout the various functions of the contract, several input parameters are not properly validated. This is specially concerning when JavaScript contracts accept undefined or nulled input parameters and do not set a default value.

The following input parameters are not being properly validated:

- `owner` in `init();`
- `token_account_id` in `init();`
- `tge_timestamp_ns` in `init();`
- `recipient` in `withdraw_unallocated();`
- `distribution_start_timestamp_ns` in `init();`

Recommendation

It is recommended to ensure all input parameters are properly validated in regards to their existence, type, and expected value, and apply default values if none are provided.

Status

The issue has been fixed in 77a14857b295b6e4a75dea151462a8eb8b07a73d.

Missing storage_withdraw() And storage_unregister()

Medium | RES-COFI-MIV05 | Business Logic | Resolved

Code Section

- merkle-claim-contract/src/contract.ts

Description

In the NEAR blockchain, contract implementations that store user state often implement storage staking mechanisms to allow users to buy in and exit the protocol by depositing NEAR to maintain their storage stake and successfully interact with it.

The storage implemented on the Merkle contract does not implement the functions `storage_withdraw()` and `storage_unregister()`, thus preventing users from unstaking their NEAR and exit the protocol should they wish to do so. This fact introduces apprehension when using ConsumerFi's protocol and may cause users to not invest time and resources into it, as their NEAR will forever be locked inside the protocol.

It should be noted that, not only does the contract solely implement storage deposit as a storage management solution, but it also allows users to deposit past the minimum amount with no chance of recovery.

Recommendation

It is recommended to implement the functions `storage_withdraw()` and `storage_unregister()` to allow users to successfully unstake their NEAR and exit the protocol.

Status

The issue has been fixed in 01efb96e245a5b958e865968ad17f9d65a930746.

Unprotected Initialization Function Can Be Frontrun

Low | RES-COFI-MIV06 | Transaction Ordering | Acknowledged

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L59
- merkle-claim-contract/src/contract.ts#L67

Description

The `init()` initialization function used to set up the protocol is unprotected. Anyone can frontrun and call this function to configure the protocol as they please. It is worth noting even after a successful exploitation the owner can still redeploy the contract to make adjustments as long as they have the key associated with the contract's `AccountId`.

Nevertheless, it is considered a best practice in the NEAR blockchain to mark the initialization functions with the decorator argument `privateFunction: true`. This enforces the predecessor `AccountId` to be equal to the contract itself, making it possible only for the contract's `AccountId` to call the function.

Recommendation

It is recommended to mark the `init()` function as a private function.

Status

The issue was acknowledged by ConsumerFi's team. The development team stated "The init function will normally called along with contract deployment on Near, and we also always only use the contract that we initialized, in order to allow admin to call init instead of contract itself.".

Missing Validation Of Duplicate investors

Low | **RES-COFI-MIV07** | Data Validation | **Resolved**

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L97-L135

Description

The function `upsert_investors()` does not validate the array `investors` against duplicate values, allowing for multiple investors to be provided in the array, where atomicity of code actions is not guaranteed and only the last investor is persisted.

Recommendation

It is recommended to validate the array to ensure that no duplicate investors are served as an input.

Status

The issue has been fixed in 52a627b2078cc3c284b7f72208ab9674b41e59a7.

Redundant Contract State Validation

Info | **RES-COFI-MIV08** | Gas Optimization | **Acknowledged**

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L71-L73
- merkle-claim-contract/src/contract.ts#L91-L93

Description

The initialization function `init()` marked with the decorator `@initialize` begins with implementing code logic to ensure that the contract is already initialized. However, this same code logic is already implemented within the NEAR SDK thanks to the use of the decorator `@initialize`.

Recommendation

It is recommended to remove redundant state verification code from the initialization function.

Status

The issue was acknowledged by ConsumerFi's team. The development team stated "Won't fix.".

JavaScript Contract Missing Schema

Info | **RES-COFI-MIV09** | Code Quality | **Acknowledged**

Code Section

- investor-claim-contract/src/investor_claim_contract.ts
- merkle-claim-contract/src/contract.ts

Description

JavaScript contracts in the NEAR blockchain, besides having the attributes that define the data the contract stores, it must also include a schema object that defines the contract's state and its types. This object is used by the SDK to correctly serialize and deserialize the contract's state.

Recommendation

It is recommended to include a Javascript schema within the definition of the class for the contract, in the following format as an example:

```
class NearContract {
  message: string = '';

  static schema = {
    "message": "string"
  };

  ...
}
```

Status

The issue was acknowledged by ConsumerFi's team. The development team stated "Won't fix.".

Redundant Code On init()

Info	RES-COFI-MIV10	Code Quality	Acknowledged

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L80-L82
- investor-claim-contract/src/investor_claim_contract.ts#L382-L384
- merkle-claim-contract/src/contract.ts#L97-L99
- merkle-claim-contract/src/contract.ts#L731-L733

Description

The function `init()` is performing the same validation of the variables `groups` and `merkle_root` as it is made in `setGroupsInternal()` and `configureDistribution()` respectively.

Recommendation

It is recommended to revise code reusability development patterns throughout the protocol, not only to improve readability, but also to maximize gas and storage efficiency on the blockchain.

Status

The issue was acknowledged by ConsumerFi's team. The development team stated "Won't fix.".

Redundant Code On assertSelf()

Info | **RES-COFI-MIV11** | Code Quality | **Acknowledged**

Code Section

- investor-claim-contract/src/investor_claim_contract.ts#L258
- investor-claim-contract/src/investor_claim_contract.ts#L281
- investor-claim-contract/src/investor_claim_contract.ts#L418-422
- merkle-claim-contract/src/contract.ts#L252
- merkle-claim-contract/src/contract.ts#L686
- merkle-claim-contract/src/contract.ts#L708
- merkle-claim-contract/src/contract.ts#L995-L999

Description

The functions `on_claim_complete()`, `on_withdraw_complete()`, `on_sync_x_token_ratio()`, marked with the decorator argument `privateFunction: true` begins by calling the function `assertSelf()`, implementing code logic to ensure that the contract is the caller of the function. However, this same code logic is already implemented within the NEAR SDK thanks to the use of the decorator argument `privateFunction: true`.

Recommendation

It is recommended to remove redundant code from the protocol.

Status

The issue was acknowledged by ConsumerFi's team. The development team stated "Won't fix.".

Unused Functions

Info | **RES-COFI-MIV12** | Code Quality | **Resolved**

Code Section

- merkle-claim-contract/src/contract.ts#L621-L624
- merkle-claim-contract/src/features/claim.ts#L552-L563
- merkle-claim-contract/src/features/claim.ts#L753-L773

Description

The following functions and modifiers were found to be unused within the system:

- `on_stake_complete()`
- `getCfiValueFromXToken()`
- `handleStakeComplete()`

Unused functions increase the complexity and readability of the smart contract's code and their inclusion should be discouraged whenever possible.

Recommendation

It is recommended to remove unused functionalities from production-ready code.

Status

The issue has been fixed in 4ea5f1c959ea8600bf8823aeb02b9feaac989055.

Redundant Code On set_distribution() And set_merkle_root()

Info | **RES-COFI-MIV13** | Gas Optimization | **Resolved**

Code Section

- merkle-claim-contract/src/contract.ts#L116-L133

Description

The functions `set_distribution()` and `set_merkle_root()` implement the same functionality and can be merged together to decrease code redundancy.

Recommendation

It is recommended to remove one or merge together both functions to save on gas usage and improve code readability and composability.

Status

The issue has been fixed in 77a14857b295b6e4a75dea151462a8eb8b07a73d.

Proof of Concepts

RES-07 Missing Validation Of Duplicate investors

main.ava.js (added lines):

```
test('duplicate investor in upsert_investors', async (t) => {
  const { worker, accounts } = t.context;
  const { root, ft, contract } = accounts;

  const now = await currentTimestamp(worker);

  const groups = [
    {
      id: 'seed',
      cliff_duration_ns: (12n * MONTH).toString(),
      vesting_duration_ns: (12n * MONTH).toString(),
    },
    {
      id: 'strategic',
      cliff_duration_ns: (12n * MONTH).toString(),
      vesting_duration_ns: (18n * MONTH).toString(),
    },
    {
      id: 'private',
      cliff_duration_ns: (12n * MONTH).toString(),
      vesting_duration_ns: (12n * MONTH).toString(),
    },
  ];

  await root.call(contract, 'init', {
    owner: root.accountId,
    token_account_id: ft.accountId,
    tge_timestamp_ns: now.toString(),
    groups,
  });

  await root.call(contract, 'upsert_investors', {
    investors: [
      {
        account_id: 'alice.test.near',
        group_id: 'seed',
        amount: (4n * ONE_TOKEN).toString(),
      },
      {
        account_id: 'alice.test.near',
        group_id: 'seed',
        amount: (5n * ONE_TOKEN).toString(),
      },
    ],
  });
```

```
  const investor = await contract.view('get_investor', { account_id:
  ↪  'alice.test.near' });
  // Only last entry is persisted
  t.is(investor.totalAllocation, (5n * ONE_TOKEN).toString());
});
```